Graph-based formalism for Machine-to-Machine self-managed communications

Cédric Eichler, Ghada Gharbi, Nawal Guermouche, Thierry Monteil, Patricia Stolf

Graph-based formalism for Machine-to-Machine self-managed communications

Cédric EICHLER[1,2,3], Ghada GHARBI[1,3], Nawal GUERMOUCHE[1,3], Thierry MONTEIL[1,3], Patricia STOLF[2,3]
[1]CNRS; LAAS; 7 avenue du Colonel Roche, F-31077 Toulouse, France
[2]IRIT; 118 Route de Narbonne, F-31062 Toulouse, France
[3]Univ de Toulouse, UPS, INSA, F-31400, UTM, Toulouse, France
{cedric.eichler, ggharbi, nawal.guermouche, thierry.monteil} @laas.fr, stolf@irit.fr

***Abstract*— Machine-to-Machine communications comprise a large number of intelligent devices sharing information and making cooperative decisions without any human intervention. To support M2M requirements and applications which are in perpetual evolution, many standards are designed, updated and rendered obsolete. Among these, arises from The European Telecommunications Standards Institute (ETSI) a promising standard for M2M communications. The ETSI M2M provides in particular a standardized framework for interoperable M2M Services. As most of its peer, this standard does not, however, address the issue of dynamic reconfiguration or provide a suitable model for the reasoning required to build self-managed M2M architectures. In our paper, we propose a graph-based approach built on top of the ETSI standard, including rules for reconfiguration management, to enforce self-management properties of M2M communications.**

Keywords-autonomic computing; dynamic reconfiguration; graph model; ETSI M2M Architecture

I. INTRODUCTION

During the last years, the exponential expansion of wireless communications devices and the ubiquity of wireless communications networks have convey to the emanation of wireless Machine-to-Machine (M2M) communications as the most promising solutions to revolutionize the future "intelligent" pervasive communications [1].

Intrinsically, M2M systems are evolution prone as applications are stopped and started; machines discovered and shut down, etc. As most of its peer, the ETSI standard focuses on protocols and communications. It does not address the issue of dynamic reconfiguration or provide a suitable model for the reasoning required to build self-managed M2M architectures. These considerations belong to the field of dynamic software architectures enabling adaptation in autonomic distributed systems, coping with new requirements, new environments, and failures. We propose in this paper a formal, component-based, bi-layered framework for modelling M2M systems. Our approach relies on a graph-based layer defined on the top of the ETSI standard. This is suitable for reasoning and handling dynamism of the corresponding system behavioural properties. We propose as well generic policies of reconfiguration, relying on graph rewriting, to enforce self-management properties.

The remainder of the paper is organized as follow: Section 2 introduces a state of the art of approaches for the description of dynamic software architectures. Section 3 introduces the approach we propose which relies on a functional and formal layer. Finally, Section 4 concludes.

II. RELATED WORKS

The description of evolving architectures cannot be limited to the specification of a unique static topology but must cover the scope of all the correct configurations. This scope characterizes an architectural style, qualifying what is correct and what is not. Naturally, once this distinction made, the question of specification of the modifications themselves arise.

Model-based approaches, proposing general-purpose modelling languages, allow handling dynamism and particularly the definition of reconfiguration rules managing the evolution on an application in run-time. They provide very intuitive and visual formal or semi-formal description of structural properties [4]. For example, designing and describing software models using UML [5] is a common practice in the software industry, providing a standardized definition of system structure and terminology, as well as facilitating a more consistent and broader understanding of the architecture [6]. Nevertheless the generic fitness of model-based approaches implies a poor means of describing specific issues like behavioural properties. Therefore, they are often coupled with description using architecture description languages [7], mapping the concepts of architecture description languages into the visual notation of UML, or other formalism [8].

Among these formalisms, graph-based methods for software modelling are appropriate for conceiving correct by design frameworks, as theoretical work in this field provides formal means to specify and check structural constraints and properties [9, 10]. Within this kind of approaches, some methods are restricted to the usage of type graphs alone [11] and suffer from a lack of expressiveness. Other works [12] are based on graph grammar, or graph rewriting system, and techniques. Graph grammars are appropriate for formal modelling dynamic structures and software architectures, and are used to specify architectural style where a graph represents a configuration. Graph rewriting rules of a

graph rewriting system have two distinct values. They are suitable for both the characterization of an architectural style as part of a rewriting system and the specification of consistency preserving reconfiguration rules.

III. THE BI-LAYERED APPROACH

This section describes the approach we propose, and particularly its layers: the functional and the formal ones. We point out that we are interested in a subset of functional properties required to enforce the management mechanisms we aim. Communication between these two layers is bi-directional. Indeed, when events, such as the discovery of a new device, arise on the functional layer, the formal layer is involved to perform reasoning and decision-making. On the other hand, whenever an action is applied consequently to a decision in the formal layer, the implication must be impacted on the functional part, such as the effective deployment of entities in the "real" world and the necessary calls for registration or announcement on the functional layer. These bi-directional updates ensure the coherence between the two layers.

A. Functional layer based on the ETSI standard

The cost of development, maintenance and research in M2M systems is increasing. To meet these challenges, the standardization is a key enabler to remove the technical barriers and ensures interoperable M2M services and networks. Many standards bodies are moving rapidly to support M2M communications requirements. They are working in defining architecture and service standards for M2M applications.

The European Telecommunications Standards Institute (ETSI) [2] has developed an end-to-end architecture for machine-to-machine communications. The ETSI standards [3] facilitate the deployment of vertical applications and the innovation across industries by exposing data and providing services. For these reasons, we have chosen the ETSI specification as a reference to model M2M systems.

ETSI has divided M2M systems into three domains:

- *Application domain*: it runs the service logic and uses M2M services capabilities accessible via an open interface. The application data is referred as resources. Resources are defined in a tree structure and handled with the RESTful style of data exchange.
- *Network domain:* it is a network technology providing connectivity between M2M devices (appliance, router, gateway, etc.).
- *M2M device domain:* it includes data end points such as sensors, smart meters, microprocessors, etc.

In M2M systems, data come from a large number of devices and are exchanged between various entities (applications) through Data Containers. These containers are used as a mediator that takes care of buffering the data. They make the exchange abstracted from the need to set direct connections and allow for scenarios where both parties in the exchange are not online at the same time. To accomplish the interaction between the distributed applications and devices (sensors, gateways, etc), the registration and the announcement of resources must be fulfilled.

B. Formal layer

Before discussing the approach we propose, we first introduce general concepts related to graph rewriting systems.

1) Graph rewriting rule and graph rewriting systems

A configuration of a system captures its state at a given time. A configuration can be modelled using attributed graphs, whose vertices specify entities (e.g., devices, applications, containers), and edges represent theirs relationships (e.g. deployment, writing, etc.).

Definition 1: (*Attributed Graph*)

An attributed graph G is defined by the tuple (V, E, ATT) where:

- V is a set of vertices
- $E \subseteq V^2$ is a set of edges
- ATT is a family of sets indexed by $V \cup E$. A set of this family is a sequence of couple (A, D_A) where A is either a constant in D_A, noted "A", or a variable, noted A, that may take any value in D_A.

An architectural style can be formalized using a graph rewriting system or graph grammar. The production rules of such systems require identifying sub-structures by the mean of morphisms. An unattributed induced sub-graph isomorphism between two graphs is defined as a homomorphism from the set of vertices of the first one to the set of vertices of the second so that if there is an edge between two vertices of the first one; there is an edge between their images in the second one and reciprocally [9].

Definition 2: (Induced sub-graph isomorphism)

There is an induced sub-graph isomorphism i between two attributed graphs G= (V, E, ATT) and G'= (V', E', ATT'), denoted $G \rightarrow G'$, if and only if there is an unattributed induced sub-graph isomorphism from (V, E) to (V', E') such as:

- $\forall v \in V$ (resp $\forall e = (\bar{v}, \bar{v}) \in E^2$), $|ATT_v| = |ATT_{h(v)}|$ (resp. $|ATT_e| = |ATT_{h(\bar{v}), h(\bar{v})}|$), (1)
- $\forall v \in V$ (resp $\forall e = (\bar{v}, \bar{v}) \in E^2$), $\forall i \in [1, |ATT_v|]$, $D^i_v = D^i_{h(v)}$, (2)
- The system of equations $S = \{A = A' \mid (\exists v \in V, \exists i \in [1, |ATT_v|], A = A^i_v \wedge A' = A^i_{h(v)}) \vee (\exists e = (\bar{v}, \bar{v}) \in E, \exists i \in [1, |ATT_e|], A = A^i_e \wedge A' = A^i_{h(\bar{v}, h(\bar{v}))})\}$ has at least one solution. (3)

Solving the system of equations S results in identifying the value of some attributes with some constants in their domains of definitions and/or with the value of some other attributes. Integrating the affectation

obtained by solving the systems refers to the update of the value of the attribute to reflect these identifications, see [13] for more information about these integrations. For genericness sake, we define the following super-patterns.

Definition 3: (Super-pattern)

A super pattern is one of the following elements:

- a vertex whose only attribute is "any", its domain of definition begin of no interest. Its attributes do not take part in the conditions (1), (2) or (3). It is only relevant in the phase where an unattributed sub-graph isomorphism is looked for.
- an attribute taking value in a subset of its domain of definition, materialized by enumerating the possibility, e.g. ("a" or "b", {"a", "b", "c"}). Such an attribute impacts the condition (3) by adding a constraint on the system of equation S.

The characterization of graph rewriting rules used in this paper is based on the Double PushOut [9] approach.

Definition 4: (Graph rewriting rule)

A graph rewriting rule is a triplet (L, K, R) where L and R are two graphs, and K -called the Inv zone- is a sub-graph of both L and R. L\K is called the Del zone and R\K is called the Add zone. A rule is applicable on a graph G if there is an induced sub-graph isomorphism i: L $\rightarrow$G and its application does not lead to the apparition of any dangling edge. Its application consists in erasing (L\K) and adding an isomorph copy of R\K integrating the affectation obtained by solving the system of equations related to i.

In this paper, graph rewriting rules are illustrated using the delta representation, where only one graph is considered. This graph is visually partitioned into three zones, from left to right the Del, Inv and Add zones.

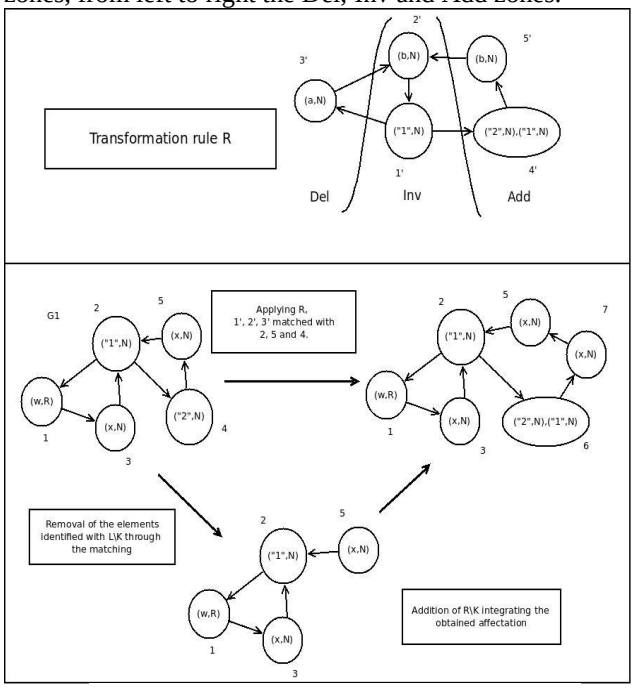

Figure 1: an example of graph transformation

Figure 1 offers an example of how transformation is handled in the previously defined approach as well as an illustration of the delta representation. To lighten the figure, the attributes of the edges have not been represented and will be all considered equals. The Del zone, for example, is composed by one vertex noted 3' and two edges (1', 3') and (3', 2'). Concerning its applicability, considering that there exists an induced sub-graph isomorphism iso such as L $\rightarrow$G1 such as $\forall\ v \in V_{G1} \setminus \text{iso}(V_K)$, $\forall\ v' \in V_L \setminus V_K$, $(v, \text{iso}(v')) \notin E_{G1} \wedge (\text{iso}(v'), v) \notin E_{G1}$, the deletion of the graph identified with Del through iso would not lead to the apparition of any dangling edge. The transformation R can be applied to G1 with the matching iso. The image of the Del zone is removed and an isomorph copy of the Add zone is then added.

Inspired from Chomsky's generative grammars [14], graph grammars are defined as follows.

Definition 5: (Graph grammar)

A graph grammar is a system <*AX; NT; T; P*>, where *AX* is the axiom, *NT* the set of the non-terminal vertices, *T* the set of terminal vertices, and *P* is the set of graph rewriting rules, also called grammars productions. An instance belonging to the graph grammar is a graph G obtained by applying a sequence of productions in *P* to AX so that there is no nt$\in$*NT* such as nt$\rightarrow$ G.

2) Characterization of the formal layer using Graph Rewriting systems

The formal layer built to reason and manage actual M2M applications is composed by a generic graph grammar. Said applications are instances of the ETSI standard for M2M architecture. In a similar fashion, management of actual M2M architectures shall rely on instances of the meta-graph grammar.

For conciseness, the information considered here are restricted to:

- The deployed devices, the kind of applications they may run, and whether they are announced or not. When two devices "see each other", i.e. they are announced to one another, the propagation delay due to the physical network through which they communicate.
- The deployed containers, on which device, and whether they are registered or announced.
- The deployed applications, on which device, their type, whether they are registered or announced, and the containers they currently use.

Consequently, the generic graph grammar is <*AX, Ø, T, P*> where:

T = {N((id,Id),(deviceType, {"Network", "Gateway", "ETSIdevice"}), (runnableAppli, appliTypes)), N((id,Id)), N((id,Id),(appliType, appliTypes))} and

P = {p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}}. For readability sake and considering that there is no ambiguity on domains of definition, they are implicit in the

following. Only the most representative productions are defined and graphically represented.

The production p_1, illustrated in figure 2, describes the initialization and the deployment of the "Network" node. The addition of a device, managed by the rule p_2, is similar and can be done at any time.

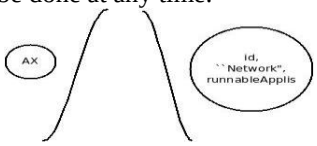

Figure 2: (p_1) Initialization

The rule p_3 illustrated in figure 3 formalizes the addition and the registration of an application. The rule p_4 modelling the deployment and registration of a container is similar and thus no represented.

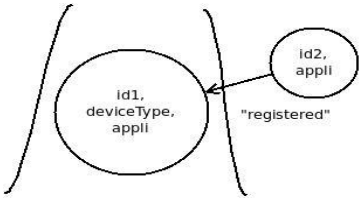

Figure 3: (p_3) Application: addition and registration

The rule p_5 presented in the figure 4 depicts the announcement of a device to the network or a gateway.

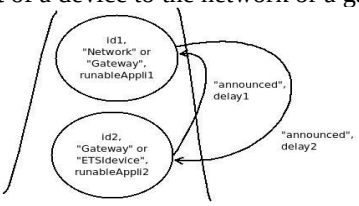

Figure 4: (p_5) Announcement of a device

The announcement of an application or a container requires the device it is deployed on to be announced as shown by the production p_6 depicted in figure 5.

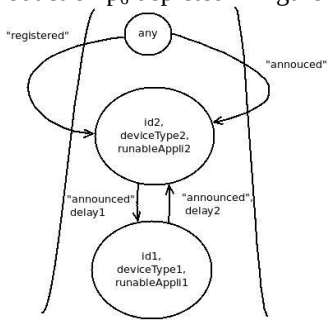

Figure 5: (p_6) Announcement of an application or a container

An application may use a container, i.e. reads and/ or writes on it, if one of the following conditions is met:

- Both are deployed on the same device, as described by the production p_7.
- The container is on an entity on which the application is announced, p_8 illustrated in figure 6.
- The application is running on an entity on which the container is announced, p_9.

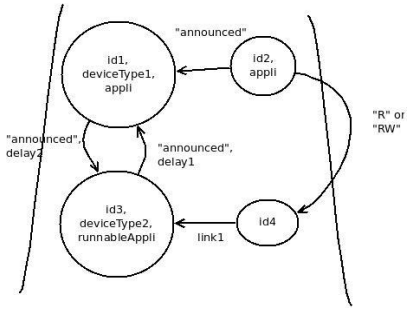

Figure 6: (p_8) An application uses a distant container

Considering this meta-graph rewriting system, when a new device on the functional layer is discovered, it triggers the application of the production p_2 with the ad-hoc attributes followed by p_3 and p_4 as many time as necessary, i.e. once by respectively applications and containers registered on the discovered device. Decisions making in the formal layer and generic algorithms for enforcing self-managed policies are presented in the next section.

IV. Enforcement of self-management policies

We now suppose the existence of a monitoring and/or an analysing routine able to throw the following events:

- there is less than x% of battery left on a device d,
- a container c has been accessed more than x times by distant applications in an interval of time t,
- an application of a certain type is needed to be seen from a device d.

Each event triggers an algorithm as described below. These algorithms use graph rewriting rules connected to the production of the meta-grammar. Actually, the application of most of them is equivalent to the application of a production or a sequence of productions of the grammar. They only differ in their applicability conditions by requiring larger patterns to be found. The suppression of a container forms a notable exception, and is based on the reversibility of productions. These facts ensure that the system stays in a state buildable with a sequence of productions, and thus the correctness of the reconfigurations. The graph representing the formal layer when an event is thrown is noted G = (V, E, ATT).

When "**a container c has been accessed more than x times by distant applications in an interval of time t"**, it should be moved to the network in order not to saturate the communication channel of the device where *c* is deployed. Every application that reads and/or writes on c is redirected to the corresponding container. These actions are described in the algorithm *migrate(idC, idD),* where idC is the identifier of c and idD the identifier of the device where the new container shall be deployed, in this case the Network.

migrate(idC, idD)
createNannounce(idC, idD)
for each induced sub-graph isomorphism i : Lredirect(idc) → G
apply graph rewriting rule redirect(idc, idNewC) w.r.t. i
update the resource tree of the application identified by i
apply graph rewriting rule destroy(idC)
update the resource tree of the device where c used to be deployed.

With *createNannounce(idC, idD)* being the process creating a new container on the device identified by idD, and making every announcement so that each application using the container identified by idC may use the new container.

createNannounce(idC, idD)
apply graph rewriting rule p_4 with id fixed idD
idNewC ← id', the id of the new container
for each induced sub-graph isomorphism i : $L_{announceD(idC, idNewC)}$ → G
apply graph rewriting rule announceD(idc, idNewC) w.r.t. i
update the resource tree of the device identified by i and the Network resource tree
apply graph rewriting rule p_6 with the isomorphism associating the super vertex with the new container.
Deploy the corresponding container and update the resource trees.

where redirect(idC,idNewC), destroy(idc), and announceD(idC,idNewC) are defined respectively in figure 7, 8 and 9. Note that the uniqueness of the induced sub-graph isomorphism, with regard to which p_4, p_6, duplicate(idC, idNewC) and destroy(idC) are applied, is ensured by the uniqueness of the identifier of the container.

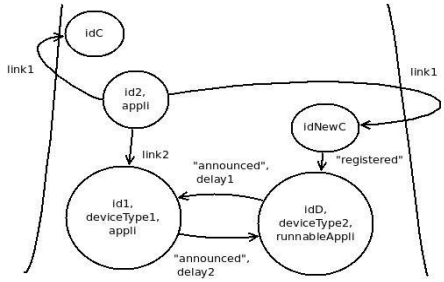

Figure 7: Redirection of an input and/ or output of an application

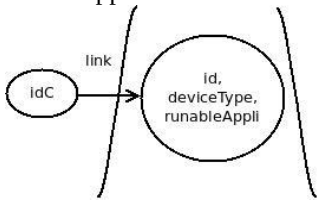

Figure 8: Suppression of the original container

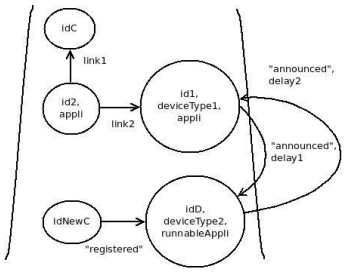

Figure 9: Announcement of a device on which an application to be redirected is deployed

The case where "there **is less than x% of battery left on a device d"**, may lead to the loss of data in the containers deployed on the device *d* whenever it will shut down due to an empty battery. In order to prevent this loss, each container deployed on d is moved elsewhere and every application that reads and/ or writes on a migrated container is redirected to the corresponding container, as conducted by the process backup(idD).

backup(idD)
for each induced sub-graph isomorphism i : G'(idD) → G
idC← the identifier of the container associated with id through i.
idTargD ← findSuitableDevice(idC)
migrate(idC, idTargD)

With G' being nothing more than a container deployed on device. findSuitableDevice is introduced at the end of this subsection.

Finally, we consider also the case where "**an application of a certain type is needed to be seen from a device d"**. In this context, the first thing to do is to look for such an application and conduct the required announcements. If there is none, such an application shall be started on a device that can run this kind of application. If there is none, such a device shall be deployed. Finally the required announcements are conducted.

lookup(type)
if there is no induced sub-graph isomorphism i : ({N((id,Id), (type, applicationTypes))}, Ø;}) → G
if p_3 is not applicable to G with appli being fixed to type
apply p_2 to G with runnableAppli fixed to type
apply p_3 to G with appli fixed to type
idA← the attribute identified with id through I or the identifier of the new application
if applicable to G apply announceDevice(idA, idD)
apply announceApp(idA, idD) to G

Where announceDevice(idA, idD) and announceApp(idA, idD) are respectively defined in figures 10 and 11.

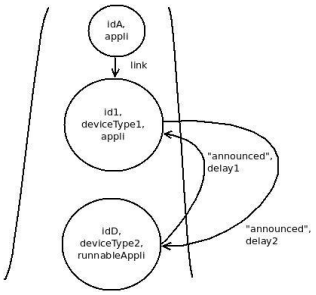

Figure 10: Required announcement of the device

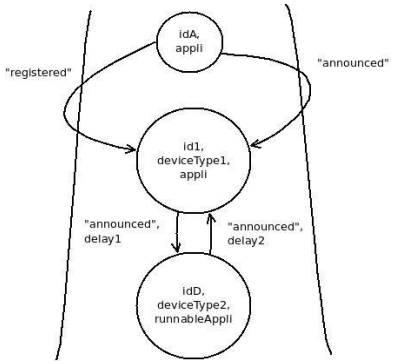

Figure 11: Required announcement of the application

The function **findSuitableDevice(idC)** returns the identifier of the device on which a container c' will be deployed in order to replace or reinforce the container c identified by idC. This implies that each application using c will use c'. We consider that the suitable device is the one minimizing the sum of the transmission delays from each application to said device. Besides, the potential targeted devices are restricted to the ones on which an application using c is deployed except the one where c is deployed, plus the network. Said set can be constructed by looking for induced sub-graph isomorphisms from R_{p9}, R_{p10}, and R_{p11} to G. It is supposed to be known and noted potential(idC), while potential_id(idC) qualifies its set of identifiers. To compute the transmission delays, we rely on Floyd-Warshall, a well-known algorithm of graph theory solving the all-pair shortest paths, with edges weighted by their attributes "delay". The function FW(G) takes a graph G and returns a function shortest,
shortest(id1, id2) being the weight of the shortest path from the vertex identified by id1 to the one identified by id2. If there is no such path, the weight is infinite.

findSuitableDevice(idC)
searchGraph ← the sub-graph of G induced by potential(idC)
shortest←FW(searchGraph)
for each i ∈ potential_id(idC)
    sumFrom(i)←$\Sigma_{id \in potentialD\text{-}id \setminus \{i\}}$ shortest(i, id)
idD← id such as sumFrom(id) = mini∈$_{potentailD\_id}$ sumFrom(id)
if sumFrom(idD) is finite return idD
else return idNetwork

IV. CONCLUSION

In this paper, we introduced a bi-layered approach for modelling and managing M2M architectures. The ETSI M2M standardization has been chosen to describe functional properties and guarantee interoperability between machines. On top of this description, we defined a graph-based generic framework ad-hoc for reasoning and handling the inherent dynamism of M2M architectures. Additionally, we have shown how this formalism may be used to enforce correct generic self-management policies of reconfiguration by defining scenarii and procedures affecting both layers to cope with new requirements and/or prevent failures. Finally, we illustrated the appropriateness of graphs and graph algorithms for decision-making.

As future work, we plan to fully define and enforce the interactions between layers. Defining a set of bidirectional actions to be performed depending on graph evolution, or events arising in the functional layer, would open the path to a formal proof of inter-consistency and, therefore, implementation.

REFERENCES

[1] S. Pandey, M-S. Mup, M-H. C, and J W. Hong, "Towards Management of Machine to Machine Networks," Network operations and Management Symposium (APNOMS), 2011 13th Asia-Pacific, vol., no., pp.1-7, 21-23 Sept. 2011

[2] "ETSI M2M", http://www.etsi.org/Website/Technologies/M2M.aspx.

[3] "ETSI M2M functional architecture technical v1.1.1", report http://www.etsi.org/deliver/etsi_ts/102600_102699/102690/01.01.01_60/ts_102690v010101p.pdf

[4] J.S. Bradbury, J.R. Cordy, J. Dingel, M, Werlinger, "A survey of self-management in dynamic software architecture specifications", Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, WOSS' 04, ACM, New York, USA, 2004, pp 28-33

[5] OMG, Unified Modeling Language Specification 2.0: Superstructure, oMG doc. formal/05-07-04 (2005).

[6] P. Selonen, J. Xu, "Validating uml models against architectural profiles", SIGSOFT Softw. Eng. Notes 28 (2003) 58–67. doi:http://doi.acm.org/10.1145/949952.940081. URL http://doi.acm.org/10.1145/949952.940081

[7] L. Broto, D. Hagimont, P. Stolf, N. de Palma, S. Temate, Autonomic management policy specification in tune, in: ACM Symposium on Applied Computing, Fortaleza, Ceara, Brazil, 2008, pp. 1658–1663.

[8] I. Loulou, A. H. Kacem, M. Jmaiel, K. Drira, Towards a unified graphbased framework for dynamic component-based architectures description in z, Pervasive Services, IEEE/ACS International Conference on 0 (2004)227234.doi:http://doi.ieeecomputersociety.org/10.1109/PERSER.2004.33.

[9] G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, World Scientific, 1997.

[10] R. Bruni, A. Bucchiarone, S. Gnesi, D. Hirsch, A. Lluch Lafuente, Graphbased design and analysis of dynamic software architectures, in: P. Degano, R. Nicola, J. Meseguer (Eds.), Concurrency, Graphs and Models, Vol. 5065 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008 pp. 37–56. doi:10.1007/978-3-540-68679-8 4. URL http://dx.doi.org/10.1007/978-3-540-68679-8 4

[11] M. Wermelinger, J. L. Fiadeiro, A graph transformation approach to software architecture reconfiguration, in: Joint APPLIGRAPH/GETGRATSWorkshop on Graph Transformation Systems (GraTra2000, 2000, pp. 200–0).

[12] D. Hirsch, P. Inverardi, U. Montanari, Modeling Software Architectures and Styles with Graph Grammars and Constraint Solving, in: P.Donohoe (Ed.), Software Architecture (TC2 1st Working IFIP Conf. on Software Architecture, WICSA1), Kluwer, San Antonio, Texas, USA, 1999, pp. 127–143.

[13] C. Chassot, K. Guennoun, K. Drira, F. Armando, E. Exposito, A. Lozes, Towards autonomous management of qos through model-driven adaptability in communication-centric systems, ITSSA 2 (3) (2006) 255–264.

[14] N. Chomsky, Three models for the description of language, Information Theory, IEEE Transactions on 2 (3) (1956) 113–124. URL http://ieeexplore.ieee.org/xpls/abs all.jsp?arnumber=1056813